IN THE SPECIFICATION:

Please replace the paragraph beginning on page 2, line 8, with the following replacement paragraph.

Portions of the disclosure of this patent document contain material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure as it appears in the Patent and Trademark Office file or records, but otherwise reserves all copyright rights whatsoever. [[Sun, Sun Microsystems, the Sun logo, Solaris, SPARC, "Write Once, Run Anywhere", Java, JavaOS, JavaStation and all Java-based]] SUN, SUN MICROSYSTEMS, the SUN logo, SOLARIS, SPARC, "WRITE ONCE, RUN ANYWHERE", JAVA, JAVAOS, JAVASTATION and all JAVA-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Please replace the paragraph beginning on page 3, line 4, with the following replacement paragraph.

To accommodate the variety of hardware systems used by clients, applications or applets are distributed in a platform-independent format such as the [[Java™]] JAVA™. class file format. Object-oriented applications are formed from multiple class files that are accessed from servers and downloaded individually as needed. Class files contain byte code instructions. A "virtual machine" process that executes on a specific hardware platform loads the individual class files and executes the bytecodes contained within.

Please replace the paragraph beginning on page 6, line 1, with the following replacement paragraph.

A hierarchy of classes can be defined such that an object class definition has one or more subclasses. A subclass inherits its parent's (and grandparent's etc.) definition.

Each subclass in the hierarchy may add to or modify the behavior specified by its parent class. Some object-oriented programming languages support multiple inheritance where a subclass may inherit a class definition from more than one parent class. Other programming languages support only single inheritance, where a subclass is limited to inheriting the class definition of only one parent class. The [[Java™]] JAVA™ programming language also provides a mechanism known as an "interface" which comprises a set of constant and abstract method declarations. An object class can implement the abstract methods defined in an interface. Both single and multiple inheritance are available to an interface. That is, an interface can inherit an interface definition from more than one parent interface.

Please replace the paragraphs beginning on page 6, line 21, with the following replacement paragraphs.

[[Java™]] JAVA™ Programming and Execution

A [[Java™]] JAVA™ program is composed of a number of classes and interfaces. Unlike many programming languages, in which a program is compiled into machine-dependent, executable program code, [[Java™]] JAVA™ classes are compiled into machine independent bytecode class files. Each class contains code and data in a platform-independent format called the class file format. The computer system acting as the execution vehicle contains a program called a virtual machine, which is responsible for executing the code in [[Java™]] JAVA™ classes. The virtual machine provides a level of abstraction between the machine independence of the bytecode classes and the machine-dependent instruction set of the underlying computer hardware. A "class loader" within the virtual machine is responsible for loading the bytecode class files as needed, and either an interpreter executes the bytecodes directly, or a "just-in-time" (JIT) compiler transforms the bytecodes into machine code, so that they can be executed by the processor.

Sample [[Java™]] JAVA™ Network Application Environment

Figure 1 is a block diagram illustrating a sample [[Java™]] JAVA™ network environment comprising a client platform 102 coupled over a network 101 to a server 100 for the purpose of accessing [[Java™]] JAVA™ class files for execution of a [[Java™]] JAVA™ application or applet.

In Figure 1, server 100 comprises [[Java™]] JAVA™ development environment 104 for use in creating the [[Java™]] JAVA™ class files for a given application. The [[Java™]] JAVA™ development environment 104 provides a mechanism, such as an editor and an applet viewer, for generating class files and previewing applets. A set of J[[Java™]] JAVA™ core classes 103 comprise a library of [[Java™]] JAVA™ classes that can be referenced by source files containing other/new [[Java™]] JAVA™ classes. From [[Java™]] JAVA™ development environment 104, one or more [[Java™]] JAVA™ source files 105 are generated. [[Java™]] JAVA™ source files 105 contain the programmer readable class definitions, including data structures, method implementations and references to other classes. [[Java™]] JAVA™ source files 105 are provided to [[Java™]] JAVA™ compiler 106, which compiles [[Java™]] JAVA™ source files 105 into compiled ".class" files 107 that contain bytecodes executable by a [[Java™]] JAVA™ virtual machine. Bytecode class files 107 are stored (e.g., in temporary or permanent storage) on server 100, and are available for download over network 101.

Client platform 102 contains a [[Java™]] JAVA™ virtual machine (JVM) 111 which, through the use of available native operating system (O/S) calls 112, is able to execute bytecode class files and execute native O/S calls when necessary during execution.

[[Java™]] JAVA™ class files are often identified in applet tags within an HTML (hypertext markup language) document. A web server application 108 is executed on server 100 to respond to HTTP (hypertext transport protocol) requests containing URLs (universal resource locators) to HTML documents, also referred to as "web pages." When

a browser application executing on client platform 102 requests an HTML document, such as by forwarding URL 109 to web server 108, the browser automatically initiates the download of the class files 107 identified in the applet tag of the HTML document. Class files 107 are typically downloaded from the server and loaded into virtual machine 111 individually as needed.

It is typical for the classes of a [[Java™]] JAVA™ program to be loaded as late during the program's execution as possible; they are loaded on demand from the network (stored on a server), or from a local file system, when first referenced during the [[Java™]] JAVA™ program's execution. The virtual machine locates and loads each class file, parses the class file format, allocates memory for the class's various components, and links the class with other already loaded classes. This process makes the code in the class readily executable by the virtual machine.

Please replace the paragraph beginning on page 9, line 12, with the following replacement paragraph.

There are a variety of applications for which [[Java™]] JAVA™ byte code or serialized objects need to be delivered to clients in a timely fashion. For example, ensuring timely delivery of byte code is essential when [[Java™]] JAVA™ byte code is used to control time aware media in a push scenario.

Please replace the paragraph beginning on page 10, line 20, with the following replacement paragraph.

A technique is needed to transform the class file (or a serialized object) into a time aware stream. If the [[Java™]] JAVA™ byte code stream were made time aware, other real time transport mechanisms like RTP could be used to transport the stream in a timely fashion. Such a scheme would facilitate the use of byte code in any multicast or broadcast scenario. This would make it possible to transmit byte code using internet protocols like User Datagram Protocol (UDP). Without such a scheme UDP is unsuitable

for delivery of byte code since UDP is an unreliable protocol (one that does not guarantee delivery of data). Transmission of byte code over an unreliable protocol could result in loss of portions of the byte code, which would prevent the byte code from executing properly.

Please replace the paragraph beginning on page 12, line 8, with the following replacement paragraph.

Embodiments of the invention enable timely delivery of [[Java™]] JAVA™ byte code in a multicast/broadcast scenario with or without a back channel. The same mechanisms can also be used for delivering (serialized) objects. Multicasting allows transmission of a specially addressed data stream to many users. With multicasting, the data stream does not need to be individually transmitted to each user. Rather, the users can subscribe to the multicasting service, for example by specifying the address of the specially addressed data stream. Broadcasting allows transmission to users without the subscription process of multicasting.

Please replace the paragraphs beginning on page 13, line 1, with the following replacement paragraphs.

Embodiments of the invention serve to make the [[Java™]] JAVA™ byte code (in a class file) "time aware". To ensure timely delivery of byte code, the appropriate deadlines are carried along with the content as time stamps to the clients. These time stamps are used in a header to facilitate such a delivery mechanism. The header is attached to the byte code to allow timely delivery of the byte code stream.

One aspect of transmission that is addressed in timely delivery of byte code is that of packet loss. Prior techniques have provided no way of recovering from a packet loss in the case of [[Java™]] JAVA™ byte code streaming. One possible approach involves retransmission of the entire class at regular intervals in the absence of a back channel. This would help to facilitate random access points in the case of media. However, this

may not be possible when the number of clients is too high or when the class (or object) is very large, making retransmission prohibitive.

Please replace the paragraph beginning on page 14, line 4, with the following replacement paragraph.

Another aspect that is addressed is that of security. To ensure the safety of the client, the byte code needs to authentic. There are a number of security schemes that can be used to ensure the authenticity of the byte code. Embodiments of the invention also accommodate the use of any security scheme within the security model in the [[Java™]] JAVA™ security APIs.

Please replace the paragraph beginning on page 15, line 3, with the following replacement paragraph.

Figure 1 is an embodiment of a [[Java™]] JAVA™ network application environment.

Please replace the paragraphs beginning on page 20, line 23, with the following replacement paragraphs.

Embodiments of the invention can be better understood with reference to aspects of the class file format. Description is provided below of the [[Java™]] JAVA™ class file format. Additional description of the [[Java™]] JAVA™ class file format can be found in Chapter 4, "The class File Format," and Chapter 5, "Constant Pool Resolution," of *The Java™ Virtual Machine Specification*, by Tim Lindholm and Frank Yellin, published by Addison-Wesley in September 1996, © Sun Microsystems, Inc.

The [[Java™]] JAVA™ class file consists of a stream of 8-bit bytes, with 16-bit, 32-bit and 64-bit structures constructed from consecutive 8-bit bytes. A single class or

interface file structure is contained in the class file. This class file structure appears as follows:

```
ClassFile {
  u4 magic;
 u2 minor_version;
 u2 major_version;
 u2 constant_pool_count;
 cp_info constant_pool[constant_pool_count-1];
 u2 access_flags;
 u2 this_class;
 u2 super_class;
 u2 interfaces.sub.-- count;
 u2 interfaces[interfaces_count];
 u2 fields_count;
 field_info fields[fields_count];
 u2 methods_count;
 method_info methods [methods_count];
 u2 attributes_count;
 attribute_info attributes[attributes_count];
 }
```

where u2 and u4 refer to unsigned two-byte and four-byte quantities. This structure is graphically illustrated in Figure 3.

Please replace the paragraphs beginning on page 22, line 12, with the following replacement paragraphs.

Magic value 301 contains a number identifying the class file format. For the [[Java™]] JAVA™ class file format, the magic number has the value 0xCAFEBABE. The minor version number 302 and major version number 303 specify the minor and

major version numbers of the compiler responsible for producing the class file.

The constant pool count value 304 identifies the number of entries in constant pool table 305. Constant pool table 305 is a table of variable-length data structures representing various string constants, numerical constants, class names, field names, and other constants that are referred to within the ClassFile structure. Each entry in the constant pool table has the following general structure:

```
cp_info {
   u1 tag;
 u1 info[];
   }
```

where the one-byte "tag" specifies a particular constant type. The format of the info[ ] array differs based on the constant type. The info[ ] array may be a numerical value such as for integer and float constants, a string value for a string constant, or an index to another entry of a different constant type in the constant pool table. Further details on the constant pool table structure and constant types are available in Chapter 4 of *The Java™ Virtual Machine Specification (supra)*.

Please replace the paragraph beginning on page 23, line 14, with the following replacement paragraph.

Access flags value 306 is a mask of modifiers used with class and interface declarations. The "this class" value 307 is an index into constant pool table 305 to a constant type structure representing the class or interface defined by this class file. The super class value 308 is either zero, indicating the class is a subclass of java.lang.Object, or an index into the constant pool table to a constant type structure representing the superclass of the class defined by this class file.

Please replace the paragraphs beginning on page 24, line 13, with the following replacement paragraphs.

The attributes count value 315 indicates the number of structures in attributes table 316. Each element in attributes table 316 is a variable-length attribute structure. Attribute structures are discussed in section 4.7 of Chapter 4 of *The Java™ Virtual Machine Specification (supra)*.

Embodiments of the invention enable timely delivery of [[Java™]] JAVA™ byte code in a multicast/broadcast scenario with or without a back channel. The same mechanisms can also be used for delivering (serialized) objects. Multicasting allows transmission of a specially addressed data stream to many users. With multicasting, the data stream does not need to be individually transmitted to each user. Rather, the users can subscribe to the multicasting service, for example by specifying the address of the specially addressed data stream. Broadcasting allows transmission to users without the subscription process of multicasting.

Please replace the paragraphs beginning on page 25, line 12, with the following replacement paragraphs.

Embodiments of the invention serve to make the [[Java™]] JAVA™ byte code (in a class file) "time aware". To ensure timely delivery of byte code, the appropriate deadlines are carried along with the content as time stamps to the clients. These time stamps are used in a header to facilitate such a delivery mechanism. The header is attached to the byte code to allow timely delivery of the byte code stream.

One aspect of transmission that is addressed in timely delivery of byte code is that of packet loss. Prior techniques have provided no way of recovering from a packet loss in the case of [[Java™]] JAVA™ byte code streaming. One possible approach involves retransmission of the entire class at regular intervals in the absence of a back channel. This would help to facilitate random access points in the case of media. However, this

may not be possible when the number of clients is too high or when the class (or object) is very large, making retransmission prohibitive.

Please replace the paragraph beginning on page 26, line 16, with the following replacement paragraph.

Another aspect that is addressed is that of security. To ensure the safety of the client, the byte code needs to authentic. There are a number of security schemes that can be used to ensure the authenticity of the byte code. Embodiments of the invention also accommodate the use of any security scheme within the security model in the [[Java]] JAVA security APIs.

Please replace the paragraph beginning on page 32, line 15, with the following replacement paragraph.

Classes are identified by an ID, which is unique to the session. This ID can be used to identify classes when it is received multiple times. This can be used by objects to identify the class of which each is an instance. [[Java™]] JAVA™ class names are used as ID's. Since these are variable length strings, the length of the string is also included in the header. The combination of the Class ID and its length (16 bits) are padded to the next 32-bit boundary.